ABSTRACT
In the last few decades, evolutionary algorithms were successfully applied numerous times for creating Boolean functions with good cryptographic properties. Still, the applicability of such approaches was always limited as the cryptographic community knows how to construct suitable Boolean functions with deterministic algebraic constructions. Thus, evolutionary results so far helped to increase the confidence that evolutionary techniques have a role in cryptography, but at the same time, the results themselves were seldom used.

This paper considers a novel problem using evolutionary algorithms to improve Boolean functions obtained through algebraic constructions. To this end, we consider a recent generalization of Hidden Weight Boolean Function construction, and we show that evolutionary algorithms can significantly improve the cryptographic properties of the functions. Our results show that the genetic algorithm performs by far the best of all the considered algorithms and improves the nonlinearity property in all Boolean function sizes. As there are no known algebraic techniques to reach the same goal, we consider this application a step forward in accepting evolutionary algorithms as a powerful tool in the cryptography domain.

CCS CONCEPTS
• Security and privacy → Mathematical foundations of cryptography;
• Computing methodologies → Evolutionary algorithms; Genetic programming;

KEYWORDS
Boolean function, Cryptography, Secondary Construction, Hidden Weight Boolean Function

1 INTRODUCTION
Evolutionary computation (EC) represents an interesting option (often as a last-resort option) for many difficult problems. Common examples include scheduling [11], cancer detection [2], and communications networks [15]. One more domain where evolutionary computation proved to be useful is the security domain. There, we can find a plethora of diverse applications, like in fuzzing [31], side-channel attacks [33], and evolution of cryptographic primitives like Boolean functions [26] and pseudorandom number generators [29]. Interestingly, it seems that the evolution of Boolean functions took most of the evolutionary computation community interest. There are several intuitive reasons for this: 1) Boolean functions are easy to encode and evolve, 2) there are multiple interesting properties, which gives numerous interesting scenarios, and 3) evolutionary computation can reach top performance for evolving Boolean function, where this performance rivals the results of the algebraic construction. Unfortunately, this also means that evolutionary computation is commonly unable to give better results than algebraic constructions (with rare exceptions like [20]), which makes this line of research interesting but not applicable in practice. Differing from this, we propose a novel application of evolutionary computation for the evolution of Boolean functions. More precisely, we call our approach EC-assisted construction of Boolean functions, where we use evolutionary computation to improve the results obtained through algebraic constructions.

Improving the results of algebraic constructions is a relevant problem as we can obtain significantly better results. This becomes especially important in cases where algebraic constructions do not provide sufficiently good results. Improving algebraic construction results is also a difficult problem as commonly, we do not know any deterministic technique to improve the results. While exhaustive search could be considered a viable option to evaluate whether improvements are possible, exhaustive search is not practically possible. Indeed, in cryptography, one commonly works with large Boolean functions with huge search space (in general, for a Boolean function of \( n \) inputs, there are \( 2^n \) possible Boolean functions).

There are no results considering evolutionary computation to help construct Boolean functions with good cryptographic properties to the best of our knowledge. Several works use evolutionary computation to produce Boolean functions to be used in algebraic constructions or to evolve algebraic constructions [22, 27]. The closest approach we found would probably be using evolutionary computation to evolve addition chains that are then used in public-key cryptography [21].

In this work, we consider a recent proposal of a generalized Hidden Weight Boolean Function (HWBF) construction [7], based
on a general construction of so-called parameterized Boolean functions [5]. This construction is efficient to implement and results in good cryptographic properties. While the properties are good, they are far that can be obtained with other constructions like Carlet-Feng [8], which are, on the other hand, very computationally complex and are then not really usable in practical stream ciphers, since these need to be lighter and faster than block ciphers and the Boolean function used as filter being the only nonlinear part of the cipher, the complexity, and speed of the whole cipher lies precisely in the Boolean function. The generalized HWBF construction can be (easily) improved by switching bits in the function’s truth table. Unfortunately, many possible positions can be swapped, and no mathematical results determine how many bits to flip (beyond the fact that if we want to improve the nonlinearity by an additive factor δ, we know that we need to change at least δ bits) or on what positions. Since random search does not give good results and exhaustive search is computationally infeasible for practical sizes, we must look for different approaches. This paper proposes an evolutionary approach where we experiment with several evolutionary algorithms and Boolean function sizes. Our results show that genetic algorithms work the best and provide significantly better nonlinearity than the Hidden Weight Boolean Function (or its generalization). We consider this to be the first work in the EC-assisted construction of Boolean functions in cryptography.

2 TECHNICAL BACKGROUND

2.1 Notation

Let n be a positive integer, i.e., n ∈ N+. The set of all n-tuples of elements in the field F2 is denoted as F2^n, where F2 is the Galois field with two elements. We denote the inner product of two vectors a and b by a · b; it equals a · b = ∑_{i=0}^{n-1} a_i b_i, with "@" being the addition modulo two (bitwise XOR). The support (supp) of a Boolean function f is the set containing the non-zero positions in the truth table representation, i.e., supp(f) = {x : f(x) = 1}. The Hamming weight w(f) of a Boolean function f equals the size of its support. For u = (u_1, ..., u_n), v = (v_1, ..., v_n) ∈ F2^n, we define the partial order on F2^n as u ≤ v if and only if u_i ≤ v_i, ∀i.

2.2 Boolean Functions

An (n, 1)-function is any mapping f from F2^n to F2, and such a function is called a Boolean function. A Boolean function f on F2^n can be uniquely represented by a truth table, which is a vector (f(0), ..., f(1)) that contains the function values of f with inputs ordered lexicographically, i.e., a ≤ b. The Walsh-Hadamard transform W_f is a unique representation of a Boolean function that measures the correlation between f(x) and the linear functions a · x [6]:

\[ W_f(a) = \sum_{x \in F_2^n} (-1)^{f(x) @ a \cdot x}. \]  

(1)

A Boolean function f is balanced if it takes the value 1 exactly the same number of times as the value 0 when the input ranges over F2^n. If the function is imbalanced, it is not suitable for usage in cryptography as one can attack its biased output.

The minimum Hamming distance between a Boolean function f and all affine functions (in the same number of variables as f) is called the nonlinearity of f. The nonlinearity \( N_f \) of a Boolean function f can be expressed in terms of the Walsh-Hadamard coefficients as [6]:

\[ N_f = 2^{n-1} - \frac{1}{2} \max_{a \in F_2^n} |W_f(a)|. \]  

(2)

The nonlinearity of a Boolean function with n inputs is bounded above as follows

\[ N_f \leq 2^{n-1} - 2^{\frac{n}{2} - 1}. \]  

(3)

This bound is usually called the Covering Radius Bound (this inequality is an equality for so-called bent functions, which exist for n even only). When n is odd, the bound given in Eq. (3) cannot be tight. Then, the maximal nonlinearity lies between \( 2^{n-1} - 2^{\frac{n+1}{2}} \) and \( 2^{n-1} - 2^{\frac{n}{2} - 1} \).

In our evolutionary assisted construction, we will concentrate on only those two properties: balancedness and nonlinearity. Still, we are interested in an additional cryptographic property called algebraic immunity (AI) [10]. Since this property is computationally expensive (e.g., one evaluation for n = 16 lasts several hours), we do not include it in the evolution process. We conducted a posteriori tests of algebraic immunity and found that the results were good (i.e., on the HWBF construction level). Finally, we note that in cryptography (and more precisely, in the design of stream ciphers), Boolean functions’ minimum size (i.e., number of variables, that is, number of input bits) with practical importance is 13 inputs [6]. For additional information about Boolean functions and their cryptology applications, we refer interested readers to [6].

2.3 Hidden Weight Boolean Functions

The Hidden Weight (Weighted) Boolean Function (HWBF) is a Boolean function in n variables defined as follows [4]:

\[ f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x_{w_f(x)} & \text{otherwise.} \end{cases} \]

The advantages of HWBF are that it has good algebraic immunity (not optimal, but at least \[ \lceil \frac{n}{3} \rceil + 1 \], it is balanced, and its output is considerably faster to compute than those of the other currently known functions having good algebraic immunity [32]. Unfortunately, HWBF has poor nonlinearity, which makes it not practical to use in cryptography. More precisely, the nonlinearity parameter equals

\[ N_f = 2^{n-1} - 2^{\left\lfloor \frac{n-2}{2} \right\rfloor}. \]  

(4)

Note that the nonlinearity given in Eq. (4) is more or less the worst nonlinearity of all known functions with optimal algebraic immunity.

3 RELATED WORK

As already discussed, evolutionary computation is commonly used to create Boolean functions with good cryptographic properties. Common examples of such works are Millan et al., where the authors apply genetic algorithms to evolve Boolean functions with high nonlinearity [17]. Millan et al. used GA, hill climbing, and a resetting step to evolve Boolean functions with high nonlinearity [18]. In their work, they considered Boolean functions up to
12 inputs. Dawson et al. investigated two-stage optimization to generate Boolean functions [9]. More precisely, they used simulated annealing and hill-climbing with a cost function motivated by the Parseval theorem to find functions with high nonlinearity and low autocorrelation. Kavut and Melek developed improved cost functions for a search that combines simulated annealing and hill climbing [13]. With that approach, the authors were able to find some functions of eight and nine inputs that have a combination of nonlinearity and autocorrelation values previously not obtained. Millan et al. proposed a new adaptive strategy for the local search algorithm for the generation of Boolean functions with high nonlinearity [19]. Hernan et al. used a multi-objective random bit climber to search for balanced Boolean functions of size up to eight inputs with high nonlinearity. [1]. Picke et al. experimented with genetic algorithms and genetic programming to find Boolean functions that fulfill several cryptographic constraints [25]. Mariot and Leporati used Particle Swarm Optimization to find Boolean functions with good trade-offs of cryptographic properties for dimensions up to 12 inputs [16]. Picke et al. conducted a detailed analysis of the efficiency of several evolutionary algorithms and fitness functions for Boolean functions with eight inputs [26]. Picke et al. used immunological algorithms to evolve highly nonlinear Boolean functions with up to 16 inputs [28].

Next, we discuss several works that used evolutionary algorithms in different combinations with algebraic constructions. Picke et al. considered an interesting approach of evolving Boolean functions that is somewhat orthogonal to ours. More precisely, the authors evolved Boolean functions in a certain number of inputs to use them to construct larger Boolean functions algebraically [27]. Picke and Jakobovic used genetic programming to evolve secondary algebraic constructions of bent (maximally nonlinear) Boolean functions [22]. Picke and Jakobovic also investigated how evolutionary algorithms can evolve algebraic constructions of S-boxes (which are vectorial Boolean functions) [23]. For a more detailed overview of evolutionary computation applications for Boolean functions in cryptography, we refer interested readers to [24].

### 4 PROBLEM DEFINITION

Recently, C. Carlet proposed a generalization of the Hidden Weight Boolean Function that allows a construction of \( n \)-variable balanced functions \( f \) from \( (n - 1) \)-variable Boolean functions \( g \) satisfying some rather light condition, see Eq. (6) below [7]. The function is defined as:

\[
 f_{g_y}(x) = (x_{w_y(x)} + 1)(g(x') + 1) + x_{w_y(x)}g(x''), \ x \in \mathbb{F}_2^n. \tag{5}
\]

Here, \( x' \) is the vector obtained from \( x \) by erasing its coordinate of index \( w_y(x) + 1 \) and \( x'' \) is the vector obtained from \( x \) by erasing its coordinate function of index \( w_y(x) \).

Function \( g \) can be any function fulfilling the property:

\[
 \forall u \in \mathbb{F}_2^n, \ g(u^{0}) \leq g(u^{1}), \tag{6}
\]

where we denote by \( u^{(j)} \) the vector obtained from \( u \) by inserting a coordinate of value \( j \) at position \( w_y(u) + 1 \) (and shifting on the right by one position all the coordinates whose indices were at least \( w_y(u) + 1 \) before the insertion). Many functions are fulfilling this property where obvious examples are monotone Boolean functions.

A Boolean function is monotone if whenever \( u \leq v \), then \( f(u) \leq f(v) \).

This generalized HWBF construction (we denote it as GHWBF) allows keeping HWBF quality of being fast to compute if function \( g \) is fast enough to compute and having good algebraic immunity while improving its nonlinearity. Note that the construction results in a function \( f \) in \( n \) variables, and to build it, we use a function \( g \) in \( n - 1 \) variables. If \( g \) is a constant function 1, then GHWBF becomes HWBF.

As already stated, there are multiple choices for the function \( g \). The current results indicate that the best nonlinearity is obtained when \( g \) is a monomial function of the form \( \prod_{i=0}^{n-1} x_i \). What is more, the best results are reached when the degree of the monomial equals \( 3 \) (i.e., has the form \( x_1x_2x_3 \)). In our experiments, we consider only the monomial functions of degree 3, and we denote the resulting function \( f_{g_y} \) as GHWBF3.

Still, this results in a large number of possible monomials one should potentially investigate. The best nonlinearity results obtained through GHWBF are given in Table 1. We also denote the values one reaches with the HWBF function and the results obtained after improving nonlinearity through random search or exhaustive search (for small \( n \)) as discussed in the next paragraphs.

<table>
<thead>
<tr>
<th>( n )</th>
<th>HWBF</th>
<th>GHWBF3</th>
<th>Exhaustive</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>22</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>44</td>
<td>46</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>88</td>
<td>94</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>9</td>
<td>186</td>
<td>194</td>
<td>–</td>
<td>216</td>
</tr>
<tr>
<td>10</td>
<td>372</td>
<td>394</td>
<td>–</td>
<td>426</td>
</tr>
<tr>
<td>11</td>
<td>772</td>
<td>802</td>
<td>–</td>
<td>862</td>
</tr>
<tr>
<td>12</td>
<td>1544</td>
<td>1628</td>
<td>–</td>
<td>1710</td>
</tr>
<tr>
<td>13</td>
<td>3172</td>
<td>3284</td>
<td>–</td>
<td>3440</td>
</tr>
<tr>
<td>14</td>
<td>6344</td>
<td>6668</td>
<td>–</td>
<td>6732</td>
</tr>
<tr>
<td>15</td>
<td>12952</td>
<td>13372</td>
<td>–</td>
<td>13674</td>
</tr>
<tr>
<td>16</td>
<td>25904</td>
<td>27158</td>
<td>–</td>
<td>27026</td>
</tr>
<tr>
<td>17</td>
<td>52666</td>
<td>54250</td>
<td>–</td>
<td>54984</td>
</tr>
<tr>
<td>18</td>
<td>105332</td>
<td>110194</td>
<td>–</td>
<td>108520</td>
</tr>
</tbody>
</table>

Table 1: The best-obtained nonlinearity results for the HWBF anf GWHBF functions. The last two columns show nonlinearity of GHWBF3 after applying random search and exhaustive search to improve the nonlinearity. The results include the first five monomials.
We consider solutions to be masks of values denoting whether we evolve a Boolean function in the form of a syntactic tree by \( g(z) \) taken at the inputs such that \( \nu \) gives computational complexity of \( 2^{|\nu|} \). One then needs to make choices on the values taken by \( g \) that can avoid contradictions.

The first fitness function uses the nonlinearity value with the goal \( \mu \) fitness \( f_1 \) = \( Nf \). 

5 EXPERIMENTAL SETUP

5.1 Encodings and Algorithms

We consider solutions to be masks of values denoting whether we flip a bit on a certain position or not. As such, we consider two intuitive solution encodings: bitstring and tree encoding. In the bitstring encoding, a bit value equal to 1 means that a bit on that position will be flipped, and a bit value equal to 0 means no change. The length of the encoding is of the same size as the length of the maximal number of positions to swap \( 2^n - 3 \), where \( n \) is the size of the Boolean function \( f \). Note that we internally map each of the individual’s bit values to the corresponding position that can be swapped in the truth table of function \( g \). We assume lexicographical ordering, where the first bit in the individual solution represents the first bit in \( g \) allowed to be swapped, etc.

The second encoding we use is tree encoding. More precisely, we evolve a Boolean function in the form of a syntactic tree by using genetic programming. The truth table of the evolved function is then applied the same way bitstring encoding is used to denote bitflips in function \( g \). Genetic programming (GP) [14] works on the population of computable expressions, where the most common form is symbolic expressions corresponding to parse trees. The building elements in a tree-based GP are functions (inner nodes) and terminals (leaves, problem variables). The terminal set consists of \( n - 3 \) input Boolean variables, denoted \( \{v_0, \ldots, v_{n-3}\} \). The function set (i.e., the set of inner nodes of a tree) should consist of appropriate Boolean operators that allow the definition of any function with \( n \) inputs. The function set used in all the experiments consists of Boolean functions OR, XOR, AND (taking two arguments), NOT (one argument), and IF (it takes three arguments and returns the second argument if the first one evaluates to ‘true’ and the third one otherwise). The application of this particular function set is based on our previous experience in applying GP to the Boolean domain.

5.2 Parameter Tuning

For all algorithms, we conduct a short tuning phase; more precisely, for GA [12], we use a 3-tournament selection, which eliminates the worst individual among three randomly selected ones. After the elimination, a new individual is produced using the crossover operator applied on the remaining two. The new individual immediately undergoes mutation subject to a defined individual mutation rate. The crossover operators are one-point and uniform crossover, performed uniformly at random for each new offspring. The mutation operator is selected uniformly at random between a simple mutation, where a single bit is inverted, and a mixed mutation that randomly shuffles the bits in a randomly selected substring. The individual mutation probability is used to select whether an individual would be mutated or not, and the mutation operator is executed only once on a given individual.

Genetic Algorithm (GA). For GA [12], we use a 3-tournament selection, which eliminates the worst individual among three randomly selected ones. After the elimination, a new individual is produced using the crossover operator applied on the remaining two. The new individual immediately undergoes mutation subject to a defined individual mutation rate. The crossover operators are one-point and uniform crossover, performed uniformly at random for each new offspring. The mutation operator is selected uniformly at random between a simple mutation, where a single bit is inverted, and a mixed mutation that randomly shuffles the bits in a randomly selected substring. The individual mutation probability is used to select whether an individual would be mutated or not, and the mutation operator is executed only once on a given individual.

Evolution Strategy (ES). We use \((\mu + \lambda)\)-ES [3], where in each generation, parents compete with offspring, and from their joint set, \( \mu \) test individuals are kept. The offspring is generated using the same mutation operators as used in the GA.

Genetic Programming (GP). In our experiments, GP uses the same steady-state tournament selection algorithm as the GA. The variation operators are simple tree crossover, uniform crossover, size fair, one-point, and context preserving crossover (selected at random), and subtree mutation [30].

5.3 Fitness Functions

The first fitness function uses the nonlinearity value with the goal of maximizing it:

\[
\text{fitness}_1 = Nf.
\] (7)

In the second fitness function, we consider the Walsh-Hadamard spectrum’s values to have more granularity and provide gradient information. Recall that the nonlinearity depends on the highest
values in the Walsh-Hadamard spectrum. Thus, observing the spectrum and minimizing the number of the highest values could also lower the final nonlinearity. Note that due to the Parseval theorem, the sum of all Walsh-Hadamard values is fixed and equals $2^n$, which means lowering some of the spectrum’s values increases some other values. Our second fitness function looks at the nonlinearity value but also the whole Walsh-Hadamard spectrum, and we aim to maximize it:

$$f_{\text{fitness2}} = Nf + \frac{2^n - \text{count}}{2^n}, \quad (8)$$

where count denotes the number of times that the Walsh-Hadamard spectrum’s largest value is encountered (since the Walsh-Hadamard values can be less than zero, we take the absolute value). As already stated, the maximal value determines nonlinearity and having that value as low as possible will increase the nonlinearity. Note that fitness function 2 gives slightly higher values than fitness function 1 and sizes 6 to 8, all algorithms value as low as possible will increase the nonlinearity. Notice that fitness function 2 improves the resulting nonlinearity values significantly different results.

### 6 RESULTS

In this section, we present the experimental results from our investigation. We consider only the first five monomials that reach the nonlinearity equal to that from $GHWBF_3$. Our results indicate very similar behavior (the maximal nonlinearity values after running the improvement experiments) for those monomials, so we believe that adding more monomials to the experiments would not bring significantly different results.

Table 2 gives the best-obtained results for the fitness function 1 and all five monomials (if existing) for all algorithms. The highest nonlinearities are given in bold font. Notice that the nonlinearity values across different monomials are rather similar. As the monomial 1 reached the highest value the most times (10 out of 13 sizes), we continue with a detailed analysis for it only.

For monomial 1, in Table 3 we give detailed results for both fitness functions and all four considered algorithms (GA, ES, GP, random search). The results are aggregated over 50 runs where we consider the maximal nonlinearity value reached in every run. The best fitness values are given in bold style. Considering fitness 1 (which equals the nonlinearity value) and sizes 6 to 8, all algorithms behave the same, and in every run, the same maximal value is obtained. For larger sizes, GA reaches the highest nonlinearity, with GP being the second best. On average, ES is the third-best algorithm, and as expected, random search performs the worst. Notice how fitness 2 improves the resulting nonlinearity values for all algorithms except random search. While ES and GP benefit from extra information provided by the fitness function already for Boolean functions with ten inputs, GA manages to reach higher nonlinearity for all Boolean function sizes from 14 to 18 inputs.

Table 3: The best obtained nonlinearities for the first five monomials, GA/ES/GP, and fitness function 1. Monomial 1 reaches the highest nonlinearity the most times. For $n = 6$, there are only four monomials satisfying the conditions for the nonlinearity of $GHWBF_3$ so there are no results to display.

<table>
<thead>
<tr>
<th>$n$</th>
<th>mon. 1</th>
<th>mon. 2</th>
<th>mon. 3</th>
<th>mon. 4</th>
<th>mon. 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$x_1x_3x_4$</td>
<td>$x_2x_3x_4$</td>
<td>$x_1x_4x_5$</td>
<td>$x_3x_4x_5$</td>
<td>$-$</td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td>22</td>
<td>24</td>
<td>24</td>
<td>$-$</td>
</tr>
<tr>
<td>8</td>
<td>$x_1x_4x_5$</td>
<td>$x_2x_4x_5$</td>
<td>$x_3x_4x_5$</td>
<td>$x_4x_5x_6$</td>
<td>$x_4x_5x_7$</td>
</tr>
<tr>
<td>9</td>
<td>104</td>
<td>104</td>
<td>102</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>10</td>
<td>$x_1x_5x_6$</td>
<td>$x_2x_5x_6$</td>
<td>$x_3x_5x_6$</td>
<td>$x_4x_5x_6$</td>
<td>$x_5x_6x_7$</td>
</tr>
<tr>
<td>11</td>
<td>438</td>
<td>438</td>
<td>438</td>
<td>434</td>
<td>438</td>
</tr>
<tr>
<td>12</td>
<td>$x_6x_7x_8$</td>
<td>$x_6x_7x_8$</td>
<td>$x_5x_6x_7$</td>
<td>$x_5x_6x_7$</td>
<td>$x_5x_6x_7$</td>
</tr>
<tr>
<td>13</td>
<td>1794</td>
<td>1794</td>
<td>1794</td>
<td>1794</td>
<td>1780</td>
</tr>
<tr>
<td>14</td>
<td>$x_1x_6x_7$</td>
<td>$x_2x_6x_7$</td>
<td>$x_3x_6x_7$</td>
<td>$x_4x_6x_7$</td>
<td>$x_5x_6x_7$</td>
</tr>
<tr>
<td>15</td>
<td>7266</td>
<td>7266</td>
<td>7270</td>
<td>7268</td>
<td>7270</td>
</tr>
<tr>
<td>16</td>
<td>$x_1x_7x_8$</td>
<td>$x_2x_7x_8$</td>
<td>$x_3x_7x_8$</td>
<td>$x_4x_7x_8$</td>
<td>$x_5x_7x_8$</td>
</tr>
<tr>
<td>17</td>
<td>14612</td>
<td>14610</td>
<td>14612</td>
<td>14612</td>
<td>14606</td>
</tr>
<tr>
<td>18</td>
<td>$x_1x_8x_9$</td>
<td>$x_2x_8x_9$</td>
<td>$x_3x_8x_9$</td>
<td>$x_4x_8x_9$</td>
<td>$x_5x_8x_9$</td>
</tr>
<tr>
<td>19</td>
<td>29162</td>
<td>29166</td>
<td>29168</td>
<td>29184</td>
<td>29196</td>
</tr>
<tr>
<td>20</td>
<td>$x_1x_9x_{10}$</td>
<td>$x_2x_9x_{10}$</td>
<td>$x_3x_9x_{10}$</td>
<td>$x_4x_9x_{10}$</td>
<td>$x_5x_9x_{10}$</td>
</tr>
<tr>
<td>21</td>
<td>58734</td>
<td>58762</td>
<td>58724</td>
<td>58732</td>
<td>58762</td>
</tr>
<tr>
<td>22</td>
<td>$x_1x_{10}$</td>
<td>$x_2x_{10}$</td>
<td>$x_3x_{10}$</td>
<td>$x_4x_{10}$</td>
<td>$x_5x_{10}$</td>
</tr>
<tr>
<td>23</td>
<td>116840</td>
<td>116796</td>
<td>116810</td>
<td>116804</td>
<td>116838</td>
</tr>
</tbody>
</table>
we depict the results for sizes 17 and 18, respectively. It is obvious
that large improvements after 40% to 50% of evaluations. ES shows
significant differences compared to the other algorithms.

Table 3: Fitness values for the first monomial. All values are rounded to one decimal place. Sizes 8 to 11 differ from the third
decimal place, while larger sizes differ from the fifth decimal place only. From the min and max columns of fitness function
2, it is easy to obtain the nonlinearity: 1) for every value that has decimal part, remove the decimal part, and 2) for every value
that is odd, subtract 1.

<table>
<thead>
<tr>
<th>n</th>
<th>min</th>
<th>max</th>
<th>average</th>
<th>stdev</th>
<th>min</th>
<th>max</th>
<th>average</th>
<th>stdev</th>
<th>min</th>
<th>max</th>
<th>average</th>
<th>stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>24</td>
<td>24</td>
<td>24.0</td>
<td>0.0</td>
<td>24</td>
<td>24</td>
<td>24.0</td>
<td>0.0</td>
<td>24</td>
<td>24</td>
<td>24.0</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>52</td>
<td>52</td>
<td>52.0</td>
<td>0.0</td>
<td>52</td>
<td>52</td>
<td>52.0</td>
<td>0.0</td>
<td>52</td>
<td>52</td>
<td>52.0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>104</td>
<td>104</td>
<td>104.0</td>
<td>0.0</td>
<td>102</td>
<td>104</td>
<td>103.9</td>
<td>0.5</td>
<td>102</td>
<td>104</td>
<td>103.8</td>
<td>0.7</td>
</tr>
<tr>
<td>9</td>
<td>216</td>
<td>216</td>
<td>216.0</td>
<td>0.0</td>
<td>214</td>
<td>216</td>
<td>214.4</td>
<td>0.8</td>
<td>212</td>
<td>216</td>
<td>214.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>436</td>
<td>438</td>
<td>437.6</td>
<td>0.6</td>
<td>426</td>
<td>430</td>
<td>428.8</td>
<td>1.1</td>
<td>426</td>
<td>432</td>
<td>428.7</td>
<td>1.5</td>
</tr>
<tr>
<td>11</td>
<td>886</td>
<td>888</td>
<td>887.9</td>
<td>0.4</td>
<td>868</td>
<td>878</td>
<td>875.1</td>
<td>2.1</td>
<td>870</td>
<td>880</td>
<td>875.4</td>
<td>2.8</td>
</tr>
<tr>
<td>12</td>
<td>1788</td>
<td>1794</td>
<td>1791.2</td>
<td>1.2</td>
<td>1738</td>
<td>1754</td>
<td>1745.6</td>
<td>4.0</td>
<td>1734</td>
<td>1758</td>
<td>1747.3</td>
<td>5.5</td>
</tr>
<tr>
<td>13</td>
<td>3608</td>
<td>3614</td>
<td>3611.2</td>
<td>1.2</td>
<td>3514</td>
<td>3560</td>
<td>3541.5</td>
<td>9.9</td>
<td>3514</td>
<td>3564</td>
<td>3547.9</td>
<td>10.4</td>
</tr>
<tr>
<td>14</td>
<td>7242</td>
<td>7266</td>
<td>7256.6</td>
<td>4.7</td>
<td>7020</td>
<td>7102</td>
<td>7061.7</td>
<td>19.3</td>
<td>7046</td>
<td>7086</td>
<td>7073.9</td>
<td>17.0</td>
</tr>
<tr>
<td>15</td>
<td>14570</td>
<td>14612</td>
<td>14594.8</td>
<td>8.7</td>
<td>14146</td>
<td>14292</td>
<td>14236.8</td>
<td>37.4</td>
<td>14144</td>
<td>14301</td>
<td>14302.1</td>
<td>69.7</td>
</tr>
<tr>
<td>16</td>
<td>29066</td>
<td>29162</td>
<td>29112.7</td>
<td>22.4</td>
<td>28172</td>
<td>28500</td>
<td>28359.4</td>
<td>86.8</td>
<td>28072</td>
<td>28744</td>
<td>28608.3</td>
<td>116.3</td>
</tr>
<tr>
<td>17</td>
<td>58574</td>
<td>58734</td>
<td>58681.7</td>
<td>25.9</td>
<td>57344</td>
<td>57748</td>
<td>57178.5</td>
<td>132.5</td>
<td>57344</td>
<td>57878</td>
<td>57429.8</td>
<td>391.4</td>
</tr>
<tr>
<td>18</td>
<td>116682</td>
<td>116840</td>
<td>116779.2</td>
<td>34.7</td>
<td>113028</td>
<td>113194</td>
<td>113091.5</td>
<td>36.1</td>
<td>113832</td>
<td>115224.0</td>
<td>391.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Fitness values for the first monomial. All values are rounded to one decimal place. Sizes 8 to 11 differ from the third
decimal place, while larger sizes differ from the fifth decimal place only. From the min and max columns of fitness function
2, it is easy to obtain the nonlinearity: 1) for every value that has decimal part, remove the decimal part, and 2) for every value
that is odd, subtract 1.

Table 3: Fitness values for the first monomial. All values are rounded to one decimal place. Sizes 8 to 11 differ from the third
decimal place, while larger sizes differ from the fifth decimal place only. From the min and max columns of fitness function
2, it is easy to obtain the nonlinearity: 1) for every value that has decimal part, remove the decimal part, and 2) for every value
that is odd, subtract 1.

Table 3: Fitness values for the first monomial. All values are rounded to one decimal place. Sizes 8 to 11 differ from the third
decimal place, while larger sizes differ from the fifth decimal place only. From the min and max columns of fitness function
2, it is easy to obtain the nonlinearity: 1) for every value that has decimal part, remove the decimal part, and 2) for every value
that is odd, subtract 1.

Table 3: Fitness values for the first monomial. All values are rounded to one decimal place. Sizes 8 to 11 differ from the third
decimal place, while larger sizes differ from the fifth decimal place only. From the min and max columns of fitness function
2, it is easy to obtain the nonlinearity: 1) for every value that has decimal part, remove the decimal part, and 2) for every value
that is odd, subtract 1.

Table 3: Fitness values for the first monomial. All values are rounded to one decimal place. Sizes 8 to 11 differ from the third
decimal place, while larger sizes differ from the fifth decimal place only. From the min and max columns of fitness function
2, it is easy to obtain the nonlinearity: 1) for every value that has decimal part, remove the decimal part, and 2) for every value
that is odd, subtract 1.
have little relevance since its truth table is only used as a bit flip mask of nonadjacent positions in the truth table of another Boolean function \( g \) of \( n - 1 \) variables. Any structure of the intermediate function of \( n - 3 \) variables that GP creates (the genotype) is disrupted in the decoding into the resulting function \( g \) (the phenotype).

(6) Unfortunately, the results for all algorithms are still rather far from the upper bound for nonlinearity.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we approach the practical problem of improving Boolean functions' nonlinearity values obtained through algebraic constructions. To this end, we develop a novel technique we call evolutionary-assisted construction of Boolean functions, and we test it on the recently proposed generalization of the Hidden Weight Boolean Function construction. Our results indicate that evolutionary algorithms (most notably, GA) significantly improve the nonlinearity values for all investigated Boolean function sizes, where the largest differences can be observed for Boolean functions with more inputs that also have practical relevance. Additionally, we
can observe that GA enables larger nonlinearity improvements over $GHWBF_3$, than the increase in $GHWBF_3$ over $HWBF$. We consider our results to be especially important as there are no other known (and practical) techniques to help improve the nonlinearity value of Boolean functions obtained through the generalized HWBF construction.

Our experiments started from construction solutions obtained through the monomial functions of degree 3 as this construction gave the best results among all the tested ones. As EA managed to improve the nonlinearity values obtained with $GHWBF_3$ considerably, it would be interesting to explore other functions $g$. We leave as open question whether it is possible to improve the nonlinearity value even more if starting with the less fit Boolean functions. Finally, we mentioned that we also require good values of algebraic immunity (AI). Since AI evaluation is expensive, we do not consider it in our fitness function in the hope that the best-obtained results will have good AI. Nevertheless, we could evaluate AI for smaller values of $n$ (e.g., up to 10) and gain insights if the increase in AI is possible.

**ACKNOWLEDGMENTS**

The research of Claude Carlet is partly supported by the Trond Mohn Foundation.
REFERENCES


